関数型プログラミング
第3回　リストと関数（1）

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

最大公約数

- 正の整数の約数とは，その数を割り切る正の数のことである．
  - 6 の約数は，1, 2, 3, 6
  - 7 の約数は，1, 7
  - 24 の約数は，1, 2, 3, 4, 6, 8, 12, 24
  - 36 の約数は，1, 2, 3, 4, 6, 9, 12, 36
- 2つの正の整数の共通の約数を公約数という．
  - 24 と 36 の公約数は，1, 2, 3, 4, 6, 12
- 2つの正の整数の公約数で最大のものを．最大公約数という．
  - 両方の整数を割り切る最も大きな整数

divisible.hs

```
divisible x d = x `mod` d == 0
```

- **d** が **x** の約数であるかどうかを調べる関数

where 節

- 関数を定義するときに，別の関数を補助的に定義して使いたい．

```
定義0 where { 定義1; 定義2; 定義3; ···· }
```

- d が x と y の公約数であるかどうかを調べる関数の定義で，divisible を使いたい．

comdiv.hs

```
comdiv x y d = divisible x d && divisible y d
          where { divisible x d = x `mod` d == 0 }
```

- where の前で改行する場合には，次の行の先頭はインデントすること．
- where で定義するのが1つだけの場合には { } は省略できる．
- divisible の本体の x と d は comdiv の引数の x と d ではないことに注意．

- where の定義で comdiv のパラメータを参照することも可能．

```
comdiv x y d = divisible x && divisible y
          where { divisible x = x `mod` d == 0 }
```

最大公約数を求める

- **x** と **y** を正の整数としたとき，**x** と **y** の公約数は，1～**x** までの数．
  - 1～**x** までの数の中の公約数で最も大きなものを分割統治で考える．
    - **x** が公約数のときには，**x** が最大公約数
    - そうでない場合には，1～**x-1** の中の最大の公約数を探せばよい

gcd1.hs

```
gcd1 x y = findComDiv x
    where { findComDiv d =
              if comDiv d then d else findComDiv (d - 1);
            comDiv d = divisible x && divisible y
              where divisible x = x `mod` d == 0 }
```

- **findComDiv d** が1～d の中の x と y の最大の公約数を求める補助関数で，分割統治で作っています．
- **comDiv d** は **d** が **x** と **y** の公約数かどうかを調べる関数で，この中でも **where** を使っています．

ユークリッドのアルゴリズム

- 世界最初のアルゴリズムと呼ばれている.
- 最大公約数を効率よく求める.
- 次の事実を用いる.
  - $\gcd(x, y) = \gcd(y, r)$
  - $\gcd(x, 0) = x$

ここで r は x を y で割った余り

$$x = q \times y + r$$

- $\gcd(x, y)$ の問題を $\gcd(y, r)$ に分割統治で解く.

gcd2.hs

```
gcd2 x y = if y == 0 then x
           else gcd2 y (x `mod` y)
```

```
% stack ghci
Prelude> :load gcd2.hs
*Main> gcd2 24 36
12
*Main> gcd2 455 273
91
```

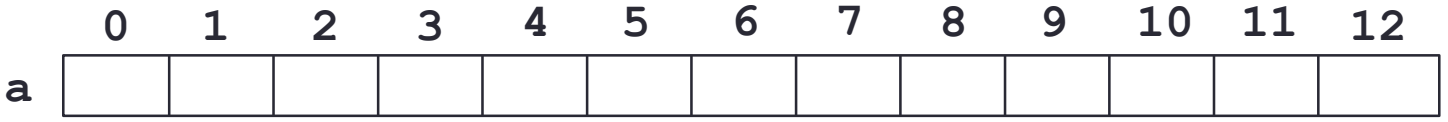
Haskellの基本的な値と型

- Haskellで取り扱うことができる基本的な値と型には以下のようなものがあります.
  - 真偽値
    - `Bool`型
  - 数値
    - `Int`型, `Integer`型, `Float`型, `Double`型
  - 文字
    - `Char`型
  - 文字列
    - `String`型 = `[Char]`型
  - タプル
    - `(a,b)`型
  - ユニット
    - `()`型
  - リスト
    - `[a]`型
  - 関数
    - `a -> b`型

配列

- たくさんのデータを塊として扱いたい.
  - クラスの学生の管理
  - 商品の管理

- 配列
  - 表のようにしてデータを管理する.
  - 前もって大きさが決まっていて, その領域を確保しておく.
  - 配列の要素は添え字を使って操作する.




```
a[2] = 3;

x = a[0]+a[5];
```

```
s = 0;
for (i = 0; i < 13; i++) {
  s = s + a[i];
}
```

リスト

- 配列
  - 利点
    - 個々の要素へのアクセスは早い.
  - 欠点
    - 要素の数が固定である.
    - 大きさの変更にはコストがかかる.

- リスト(単方向リスト)
  - 要素をつなげていくことで管理

  - 利点
    - 要素の数が可変である
  - 欠点
    - 要素は前から順番にアクセスする必要がある.
    - ランダムなアクセスは苦手

Haskellのリスト

- 同じ型の値をならべたもの
  - 色々な型のデータをまぜることはできません.
  - 単方向リストのため前から後ろにたどることしかできません.
  - 逆にはたどれません.

- リスト型
  - `[a]`
  - `[Int]`
  - `[Bool]`
  - `[Float]`
  - `[[Int]]`

- リストの例
  - `[1, 2, 3]`
  - `[True, False]`
  - `[1.5, 2.3, 4.6, 7.8]`
  - `[[1, 2, 3], [4], [5, 6], [7, 8, 9, 10, 11]]`
  - `[]`

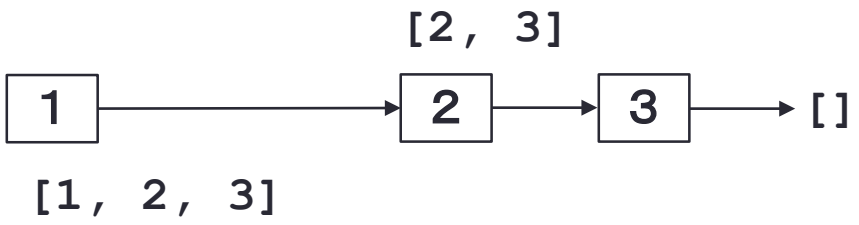
`[1, 2, 3]`

`[True, False]`

空リスト

「:」演算子

- `(:) :: a -> [a] -> [a]`
  - `x : xs`
  - リスト `xs` の先頭に `x` を追加したリストを返す



- 例
  - `1 : [2, 3]` → `[1, 2, 3]`
  - `4 : []` → `[4]`

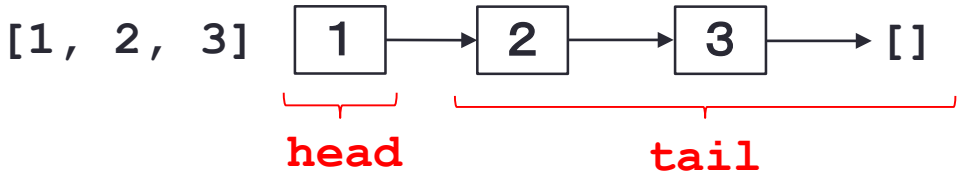
- 「:」演算子は右結合
  - `[1, 2, 3] = 1 : 2 : 3 : []`

head と tail 関数

- **head :: [a] -> a**
  - **head xs**
  - リスト **xs** の先頭要素を返す

リスト
- コンストラクタ:「:」演算子
- デストラクタ:**head** と **tail**

- 例
  - **head [1, 2, 3]** → **1**
  - **head [True, False]** → **True**
  - **head [1.5]** → **1.5**



- **tail :: [a] -> [a]**
  - **tail xs**
  - リスト **xs** から先頭要素を取り除いた残りを返す

- 例
  - **tail [1, 2, 3]** → **[2, 3]**
  - **tail [True, False]** → **[False]**
  - **tail [1.5]** → **[]**

空リストに対して **head** や **tail** はエラー

null 関数

- `null :: [a] -> Bool`
  - `null xs`
  - リスト `xs` のが空リストなら `True`，そうでないなら `False`

- 例
  - `null [1, 2, 3]` → `False`
  - `null []` → `True`
  - `null [True]` → `False`

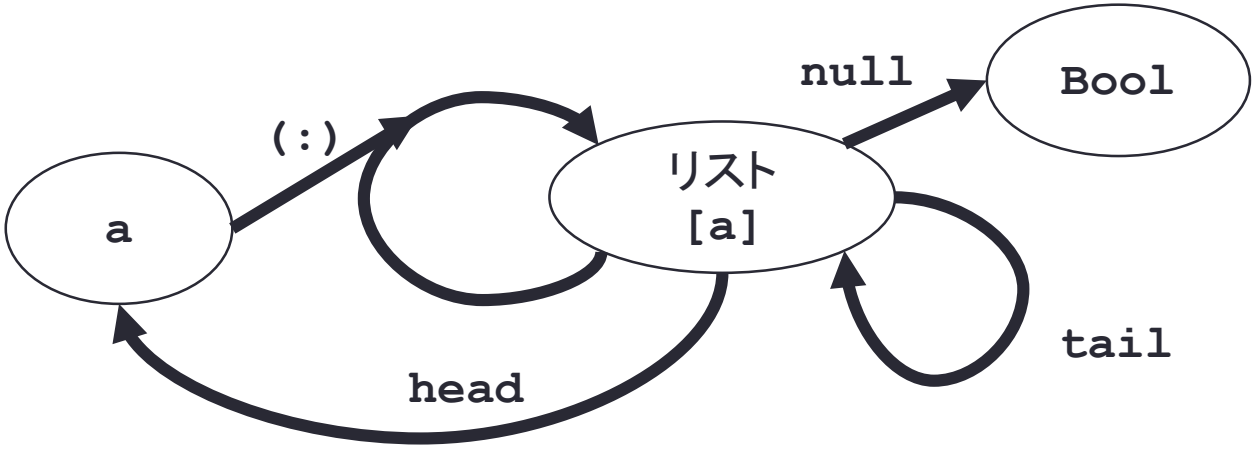
onlyone.hs

```
onlyone xs = not(null xs) && null(tail xs)
```

  - 要素が一つだけであることを確認する

リストに関する関数

関数名	適用例	意味
`(:)::a->[a]->[a]`	`x : xs`	リスト`xs`の先頭に`x`を追加したリストを返す
`head::[a]->a`	`head xs`	リスト`xs`の先頭の要素を返す
`tail::[a]->[a]`	`tail xs`	リスト`xs`から先頭の要素を取り除いたリストを返す
`null::[a]->Bool`	`null xs`	リスト`xs`が空リストなら`True`を返す



練習問題3－1

- 与えられたリストの2番目の要素を返す関数 **head2** を定義しなさい.

head2.hs

```
head2 xs = ....
```

```
% stack ghci
Prelude> :load head2.hs
*Main> head2 [1,2,3]
2
```

- **head** と **tail** を組み合わせればよい.

リストの分割統治

- 自然数の分割統治
  - より小さい数（簡単に解ける数）にする

fact.hs

```
fact n = if n == 0 then 1 else n * fact(n - 1)
```

- リストの分割統治
  - より短いリストにする
  - 空リスト `[]` のときの解を考える
  - 空リストでない場合には，`tail` で短くして再帰的に適用する

練習問題3−2

- 与えられたリストの長さを求める関数 **leength** を定義しなさい.
- 分割統治
  - 空リストの長さは 0
  - **tail** で短くしたリストの長さが分かれば，それに 1 加えれば長さが分かる

leength.hs

```
leength xs = if null xs then 0 else leength(tail xs) + 1
```

```
% stack ghci
Prelude> :load head2.hs
*Main> leength [1,2,3]
3
```

練習問題3－3

- 整数のリストが与えられたときに，合計を計算する **suum** を定義しなさい．

- 分割統治
  - 空リストの合計は 0
  - **tail** で短くしたリストの合計が分かれば，それに最初の整数加えれば合計が計算できる

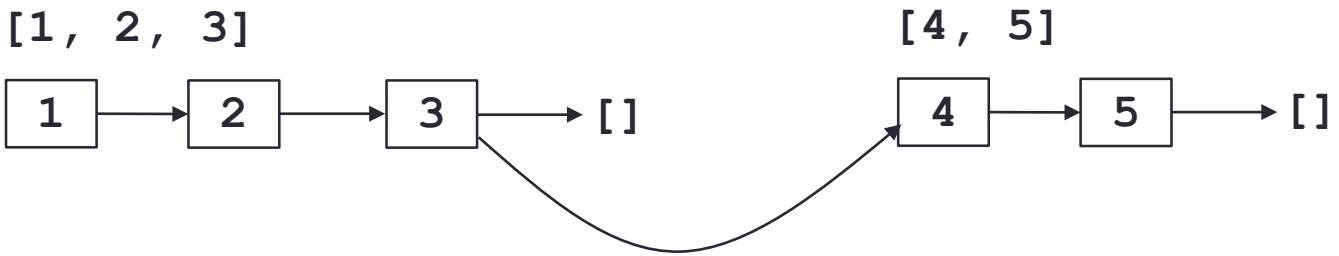
suum.hs

```
suum xs = if null xs then 0 else head xs + suum(tail xs)
```

```
% stack ghci
Prelude> :load head2.hs
*Main> suum [1,2,3]
6
```

リストの結合

- 2つのリストを結合する **append** 関数を定義しなさい.
  - **[1, 2, 3]** と **[4, 5]** を結合すると **[1, 2, 3, 4, 5]** となる.



- 分割統治: **append xs ys**
  - **xs** の先頭を取り除いたものを **ys** と結合して, その先頭に **xs** の先頭をつければよい.
  - **xs** が空リストなら結合した結果は **ys** である.

append.hs

```
append xs ys = if null xs then ys
               else (head xs):(append(tail xs) ys)
```

練習問題3－4

- リストを逆順に並び替える **rev** 関数を定義しなさい.
  - `rev [1, 2, 3]` → `[3, 2, 1]`
  - `rev [True, False]` → `[False, True]`
  - `rev []` → `[]`

rev.hs

```
rev xs = if null xs then []
         else ...
```

- 分割統治：**rev xs**
  - **xs** の先頭を取り除いた部分を逆順に並び替えることができれば，その最後に **xs** の先頭をつければよい.
  - **xs** が空リストならば，そのまま.

練習問題3－5

- 偶数 n が与えられたとき，[2， 4， 6， ....， n] のように，n 以下の偶数のリストを返す関数 geneven を定義しなさい．

geneven.hs

```
geneven n = if n == 0 then ... else ...
```

```
% stack ghci
Prelude> :load geneven.hs
*Main> geneven 10
[2, 4, 6, 8, 10]
```

- 分割統治
  - n-2 の geneven ができれば，その最後に n をつけ足せばよい．
  - n が 0 のときは空リストでよい．

練習問題3−6

- 与えらえた整数のリストのそれぞれの要素を2乗したリストを返す `sl` 関数を定義しなさい.
  - `sl [1, 2, 3]` → `[1, 4, 9]`
  - `sl [5, 8]` → `[25, 64]`
  - `sl []` → `[]`

`sl.hs`

```
sl xs = if null xs then [] else ...
```

```
% stack ghci
Prelude> :load sl.hs
*Main> sl [1, 8, 3]
[1, 64, 9]
```

- 分割統治
  - 1つ短いリストの `sl` に分割して統治する.

練習問題3－7

- **n** が偶数の場合には，2 から **n** までの偶数の2乗の和を計算し，n が奇数の場合には，1 から **n** までの奇数の2乗の和を計算する関数 **bs** を定義しなさい．
  - nが偶数のとき，$2^2+4^2+6^2+\cdots+n^2$
  - nが奇数のとき，$1^2+3^2+5^2+\cdots+n^2$

bs.hs

```
bs n = ...
```

- 方策1
  - これまでの練習問題を組み合わせて解く．
  - 奇数が偶数かで **genodd** と **geneven** を使い分ける．
- 方策2
  - **n** について分割統治で考える．

```
% stack ghci
Prelude> :load bs.hs
*Main> bs 10
220
*Main> bs 9
165
```

まとめ

- リスト
  - コンストラクタ
    - `(:) :: a->[a]->[a]`
  - デストラクタ
    - `head :: [a]->a`
    - `tail :: [a]->[a]`
  - 条件関数
    - `null :: [a]->Bool`
- リストに関する分割統治
- 構文
  - `where` 節